All Heap No Leaks - accu.org

Overload Journal #60 - Apr 2004 + Programming Topics Author: Paul Grenyer

The use of new and delete in C++ can cause problems for both novice and expert developers alike. Using new without a corresponding delete results in a memory leak. Some languages such as C#, Java and Python provide managed objects that can be created on the managed heap leaving deletion to the garbage collector.

Smart pointers allow similar behaviour to be achieved in C++, with the added bonus that they provide deterministic destruction of the object to which they point. However, the user still has to know that they should be using smart pointers and many don't. In this article I am going to look at a way of writing C++ objects that can only be created on the heap, and that must be managed by (memory management) smart pointers.

Creating Objects on the Heap

There are a number of situations in which it makes sense to force objects to be created on the heap. I recently developed a thin database access layer for use with my cross-platform test framework [Aeryn]. The sole purpose of the layer is to allow databases to be created and dropped before and after tests. Although Open Database Connectivity (ODBC) is available on both Windows and Linux (my initial target platforms), I decided to allow the use of ActiveX Data Objects (ADO) on Windows as it sometimes provides better performance.

ODBC and ADO are very different. ODBC is implemented as a C based API and ADO is implemented as a family of COM objects. However, both use a connection string to connect to a database and allow the execution of strings containing SQL statements. Therefore both types of database connection can be used via a common abstract base class.

Like many other types of resources a database connection is expensive and the most expensive operation is generally the initial connection to the database. In an ideal situation an application would have a single database connection object that is shared by all of the objects in the application that require database access. Connection to the database would be put off until absolutely necessary (lazy evaluation), and maintained for as long as possible. This results in the connection being created in one part of the application and (potentially) destroyed in another.

A common interface to one or more concrete objects, and the lifetime requirements of a database connection object, make it an ideal candidate for heap creation. One of the most likely scenarios is that the appropriate database connection object would be created at the highest possible level via a Factory Method [Gamma-] that takes the connection string as a parameter. The database connection object is then passed to objects in the application that need to use it [Gamma-], via a reference counted smart pointer [Henney2]). The first object to attempt to execute a SQL statement causes a connection to the database to be made, and then the connection is maintained throughout the lifetime of the database connection object.

As the database connection object is created via a factory method, it follows that it should be destroyed via a Disposal Method [Henney3]. The disposal method can be passed to the reference counted smart pointer and used to destroy the database connection object instead of delete. This is an example of the Resource Acquisition Is Initialisation [Henney4] idiom provided by smart pointers.

The use of factory and disposal methods gives the writer of the database connection object complete control over how and where the object is created.

Background

In Item 27 of More Effective C++ [Meyers] Scott Meyers discusses ways of restricting objects so that they can only be created on the heap. Meyers explains that sometimes a developer wants to create objects that can destroy themselves by calling delete this. If an attempt was made to create such an object on the stack the effects could be catastrophic, so it makes sense to enforce heap creation.

Equally there are times when a developer wants to prevent an object from being created on the heap. For example local (or automatic) variables are unlikely to require heap creation. Meyers also discusses preventing objects from being created on the heap. This can be achieved by declaring `operator new` and `operator new[]` private. Meyers also suggests that unless there is a compelling reason not to, `operator delete` and `operator delete[]` should also be made private.

```
namespace Meyers {
  class NotOnHeap {
  private:
    static void* operator new(size_t);
    static void operator delete(void*);
    static void* operator new[](size_t);
    static void operator delete[](void*);
  };
}
```

Meyers forces heap-only creation by making the object's destructor private (or protected if you want to be able to inherit from the object) and implementing a disposal method. For example:

```
namespace Meyers {
  class HeapOnly {
  private:
    ~HeapOnly() {}
  public:
    // Disposal Method
    void Destroy() { delete this; }
  };
}
```

If an attempt is made to create this object on the stack:

```
int main() {
  using namespace Meyers;
  HeapOnly heapOnly;
}
```

the compiler will issue an error. However, if the object is created on the heap:

```
int main() {
  using namespace Meyers;
  HeapOnly* pHeapOnly = new HeapOnly;
  pHeapOnly->Destroy();
}
```

the compiler is perfectly happy (gcc 3.2.3 [gcc] issues a warning about `HeapOnly` only having a private destructor and no friends).

In terms of forcing objects to be created on the heap, Meyers's example does exactly what is needed. However, it does have one drawback. The developer using the heap-only object must explicitly call `Destroy` in order to prevent a memory leak. Wouldn't it be better if when the pointer pointing to the heap-only object went out of scope it destroyed the object automatically? After all that is what this article is about! Enter smart pointers!

Managing Heap-Only Objects with Smart Pointers

With the exception of developing this managed heap-only object, I don't remember the last time I explicitly used the `delete` keyword. Every time I create an object on the heap I use a smart pointer to prevent a memory leak. Generally, I use Boost's [boost] `scoped_ptr` and `shared_ptr`, although there are other smart pointers available, including the reference counted smart pointer described by Meyers at the end of More Effective C++ [Meyers].

So what happens if `shared_ptr` is used with Meyers's heaponly object?

```
int main() {
  using namespace Meyers;
  boost::shared_ptr<HeapOnly> pHeapOnly(new HeapOnly);
}
```

The compiler reports an error because `shared_ptr` cannot call `HeapOnly`'s private destructor. Herb Sutter discusses some ways of getting around this in his CUJ [cuj] article Befriending Templates [Sutter]. However, I prefer a different approach suggested by an `accu-general` regular, James Slaughter, which requires some changes to Meyers's `HeapOnly` object.

So, let's start with a new object, a managed object rather than a heap-only object:

```
class Managed {
private:
  ~Managed() {}
public:
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
};
```

`shared_ptr` can take a second constructor argument, which is a function pointer used to destroy the object. To accommodate this, the managed object's `Destroy` member function is now static and has a single parameter which is a pointer to the managed object. The pointer is used to delete the object instead of `delete this`. The lifetime of the managed object can now be fully managed by the shared pointer like this:

```
int main() {
  boost::shared_ptr< Managed > pManaged(new Managed, Managed::Destroy);
}
```

The above syntax isn't as nice as it could be and the developer using the managed object must still make a conscious decision to use the smart pointer and must also know that `shared_ptr`'s constructor can take a second parameter. Wouldn't it be better if the developer could only use the smart pointer? Some would argue that restricting a developer is bad. I agree in a lot of instances, but not this one, when the aim is to have a heap-based managed object that cannot leak memory.

So, how can a developer be forced to use a smart pointer? The answer is to make all the managed object's constructors, including the copy-constructor, private (or protected if the you want to inherit from the object) and provide a static factory method that returns a smart pointer (obviously if the managed object has a number of different constructors then a factory method that takes the appropriate parameters must be added for each):

```
class Managed {
public:
  typedef boost::shared_ptr< Managed > SmartPtr;
  // Factory Method
  static SmartPtr Create() {
    return SmartPtr(new Managed, Destroy);
  }
private:
  Managed() {}
  ~Managed() {}
  // Disposal Method
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
  Managed(const Managed&);
};
```

An instance of the managed object can then be created in the following way:

```
int main() {
  Managed::SmartPtr pManaged = Managed::Create();
}
```

Although we now have a managed C++ object, the solution still isn't as versatile as it could be. What if the developer using the managed object is not able to use Boost (possibly for some commercial reason) or would like to use a custom smart pointer or a smart pointer that offers thread safety? Any type of memory management smart pointer can be used with the managed object, as long as it supports assignment and can take a disposal method as a second constructor parameter. The introduction of a template template parameter [Vandervoorde-] allows smart pointers to be interchanged easily.

```
template< template< class > class SmartPtrT >
class Managed {
public:
  typedef SmartPtrT< Managed > SmartPtr;
  static SmartPtr Create() {
    return SmartPtr(new Managed, Destroy);
  }
private:
  Managed() {}
  ~Managed() {}
  static void Destroy(Managed* pManaged) {
    delete pManaged;
  }
  Managed(const Managed&);
};
```

The syntax used to create the parameterised managed object can be simplified by introducing a typedef:

```
  int main() {
    typedef Managed< boost::shared_ptr > Managed;
    Managed::SmartPtr pManaged = Managed::Create();
  }
```

Ok, so now we have a managed object that can be used with different smart pointers. However, this solution requires a lot of extra code to be added to each new managed class written. Let's have a look at some boilerplate code that can be used to simplify the conversion of regular objects into managed objects.

Managed Object Boilerplate Code

First we need a regular object to convert to a managed object. An abstract base class pointer usually points to an object created on the heap. Therefore in this example I am going to use the simple class hierarchy that the thin database access layer described above provides, as it consists of an interface class with a single pure virtual member function, and a concrete class that implements the base class's member function:

```
  class IConnection {
  public:
    virtual ~IConnection() {}
  public:
    virtual void ExecuteSQL(const std::string& sql) = 0;
  };

  class Connection : public IConnection {
  public:
    explicit Connection(const std::string& connectionString) {
      std::cout << "Connection String: "
                << connectionString << "\n";
    }
    virtual ~Connection() {
      std::cout << "Disconnecting\n";
    }
    virtual void ExecuteSQL(const std::string& sql) {
      std::cout << "Execute: " << sql << "\n";
    }
  };
```

`IConnection` cannot be created on the stack or the heap, as it has a pure virtual member function. Connection can be created on the stack or the heap, as it has the compiler generated default public constructor and destructor (see item 45 of Scott Meyers' Effective C++ [Meyers2]), and no pure virtual member functions.

For `Connection` to be fully managed it must be prohibited from being created on the stack or on the heap, other than via a factory method. Preventing heap creation is easy, as discussed above; all that needs to be done is to define private or protected implementations of `operator new` and `operator new[]`. The factory method can then use these implementations to create an instance of `Connection` on the heap.

There are two ways of preventing `Connection` from being created on the stack: As discussed above, it can be given private or protected constructors and destructors or a pure virtual member function.

As the intention is to create boilerplate code `Connection` itself should be modified as little as possible. Inheriting from a managed base class is a simple way of adding (and documenting) managed behaviour. The following `ManagedBase` class defines a protected `operator new` and `operator new[]` (with their corresponding `delete` operators), disposal method, virtual destructor to ensure proper deletion, and pure virtual member function:

```
class ManagedBase {
private:
  template< typename ConcreteT, typename InterfaceT,
            template< class > class SmartPtrT >
  friend class Managed;
  class OnlyAvailableToManaged {};
  virtual void ForceUseOfManaged (const OnlyAvailableToManaged&) const = 0;
  template< template< class > class SmartPtrT,
            class InterfacePtrT,
            class ConcretePtrT,
            class ConcreteTypeT >
  friend class SmartPtrPolicyTraits;
protected:
  virtual ~ManagedBase() {}
  static void Destroy(const ManagedBase* pManagedBase) {
    delete pManagedBase;
  }
  static void* operator new(size_t size) {
    return ::operator new(size);
  }
  static void operator delete(void *pObject) {
    ::operator delete(pObject);
  }
  static void* operator new[](size_t size) {
    return ::operator new[](size);
  }
  static void operator delete[](void *pObject) {
    ::operator delete[](pObject);
  }
};
```

In practice there is nothing to stop the creator of `Connection` from declaring its constructors and destructors public, and therefore negating that method of preventing stack creation. If a pure virtual member function is used, which in its simplest form would have a return value of void and exactly zero parameters, the creator of `Connection` could allow stack creation by overriding it and giving it a body.

One solution to the problem presented by using a pure virtual member function to prevent stack creation was suggested in a private email from Mikael Kilpelainen. If the pure virtual member function (`ForceUseOfManaged`) has a parameter (`OnlyAvailableToManaged`) of a type that is private to the class in which the pure virtual member function is declared, it is possible to restrict the classes that can override the pure virtual member function to friends of the class in which it is declared.

Therefore, the only way that the creator of `Connection` can override the pure virtual member function declared in `ManagedBase` is to make `Connection` a friend of `ManagedBase`. This cannot be done without modifying `ManagedBase`. The knock-on effect is that another class must be provided that derives from `Connection` and is a friend of `ManagedBase`. This class is discussed next.

`ManagedBase` lacks the necessary factory method. Although it is possible for a base class to create an instance of its subclass [Henney5] a factory method is not present in `ManagedBase`. The factory method is a member of another class, `Managed`:

```
template< typename ConcreteT,
          typename InterfaceT,
          template< class > class SmartPtrT >
class Managed : private ConcreteT {
public:
  typedef SmartPtrT<InterfaceT> InterfacePtr;
  typedef SmartPtrT<ConcreteT> ConcretePtr;
private:
  ~Managed() {}
public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
    ::Initialise(new Managed);
  }
};
```

`Managed` has a private destructor to prevent stack creation. As `Managed`'s factory method will create an instance of `Connection` (`ConcreteT`) and return a smart pointer (`SmartPtrT`) of type `IConnection` (`InterfaceT`), all three must be specified as template parameters.

Managed inherits from `ConcreteT` so that it can override and define a body for the pure virtual function (`ForceUseOfManaged`) defined in `ManagedBase`. The smart pointers for the types `ConcreteT` and `InterfaceT` are typedef'd for convenience.

The factory method (`Create`) uses traits [Myers] & [Vandervoorde-] to enable initialisation code for different smart pointers to be added easily. The factory method creates an instance of `Managed` on the heap, and passes a pointer to it to the trait Initialise function. The Initialise function returns a copy of the initialised smart pointer, which is then returned by the factory method.

When calling the trait `Initialise` function the smart pointer type must be specified so that the compiler knows which trait to use. The interface type and concrete smart pointer type and the concrete type must also be specified as these are used by the `Initialise` function. The traits template is defined as follows:

```
template< template< class > class SmartPtrT,
          class InterfacePtrT,
          class ConcretePtrT,
          class ConcreteTypeT >
class SmartPtrPolicyTraits;
```

A partial specialisation [Vandervoorde-] is used to define the way in which Boost's `smart_ptr` is initialised:

```
template< class InterfacePtrT,
          class ConcretePtrT,
          class ConcreteTypeT >
class SmartPtrPolicyTraits< boost::shared_ptr,
                            InterfacePtrT,
                            ConcretePtrT,
                            ConcreteTypeT > {
public:
  static InterfacePtrT Initialise(ConcreteTypeT* pConcrete) {
    return ConcretePtrT(pConcrete, &ManagedBase::Destroy);

  }
};
```

`Initialise` needs to be able to access `Destroy`, the protected static member function defined in `ManagedBase`. One way to allow this would be to make `Destroy` public, but then it could be called from anywhere. `Destroy` should only be called by the specified smart pointer. The alternative solution is to declare `SmartPtrPolicyTraits` a friend of `ManagedBase`. The required friend template is shown in the definition of `ManagedBase` above, but here it is again:

```
class ManagedBase {
protected:
template< template< class > class SmartPtrT,
          class InterfacePtrT,
          class ConcretePtrT,
          class ConcreteTypeT >
  friend class SmartPtrPolicyTraits;
  ...
};
```

All the boilerplate code is now in place. In order to make `Connection` a managed object it must be modified to inherit from `ManagedBase`:

```
class Connection : public ManagedBase,
                   public IConnection {
public:
  explicit Connection(const std::string& connectionString) {
    std::cout << "Connection String: "
              << connectionString << "\n";
  }
  ...
};
```

There is still one outstanding problem. `Connection`'s constructor takes a single argument and `Managed` does not have the appropriate constructor or factory method. There are at least two possible solutions to this problem; each has its advantages and disadvantages.

The first is to create a partial template specialisation of `Managed` specifically for `Connection`, which has the appropriate constructor and factory method:

```
template< typename InterfaceT,
          template< class > class SmartPtrT >
```

```
class Managed< Connection,
               InterfaceT,
               SmartPtrT >
         : private Connection {
public:
  typedef SmartPtrT<InterfaceT> InterfacePtr;
  typedef SmartPtrT<Connection> ConcretePtr;
private:
  explicit Managed(const std::string& connectionString)
        : Connection(connectionString) {}
  ~Managed() {}
public:
  static InterfacePtr Create(const std::string& connectionString) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 Connection >
              ::Initialise(new Managed(connectionString));
  }
};
```

The advantage with this solution is that you can control how the constructor parameter is passed to `Create` and on to `Managed`'s constructor. For example, if for some reason `Connection` modified `connectionString`, the partial template specialisation allows the parameter to be specified as and passed by (non-const) reference. The disadvantage of this solution is that it is a lot of extra code, as it is effectively a reimplementation of `Managed`.

The second solution is to give `Managed` a number of templated constructors and `create` function pairs. For example:

```
template< typename ConcreteT,
          typename InterfaceT,
          template< class > class SmartPtrT >
class Managed : private ConcreteT {
  ...
private:
  explicit Managed() : ConcreteT() {}

  template< typename A >
  explicit Managed(const A& a)
      : ConcreteT(a) {}

  template< typename A, typename B >
  explicit Managed(const A& a, const B& b)
      : ConcreteT(a, b) {}

  ...

public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
               ::Initialise(new Managed);
  }

  template< typename A >
  static InterfacePtr Create(const A& a) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
               ::Initialise(new Managed(a));
    }

  template< typename A, typename B >
  static InterfacePtr Create(const A& a, const B& b) {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
               ::Initialise(new Managed(a, b));
  }
};
```

The advantage is that extra code is not needed for objects that have different numbers of constructor parameters, assuming that none of the objects have a number of constructor parameters greater than the constructor and `Create` pair with the highest number of parameters. The disadvantages are that you cannot control how the parameters are passed to `Create` and onto the constructor, as you can by using the partial template specialisation and that you cannot use an object that has more constructor parameters than the constructor and `Create` pair with the highest number of parameters unless you modify `Managed`. There is, however, a workaround for the second disadvantage using the boost [boost] `PreProcessor` library, shown to me by Paul Mensonides:

```
  #include <boost/preprocessor/arithmetic/inc.hpp>
  #include <boost/preprocessor/repetition/enum_binary_params.hpp>
  #include <boost/preprocessor/repetition/enum_params.hpp>
  #include <boost/preprocessor/repetition/repeat.hpp>

  #ifndef MAX_ARITY
  #define MAX_ARITY 1
  #endif

  template< typename ConcreteT,
            typename InterfaceT,
            template< class > class SmartPtrT >
  class Managed : private ConcreteT {
  public:
    typedef SmartPtrT<InterfaceT> InterfacePtr;
    typedef SmartPtrT<ConcreteT> ConcretePtr;
  private:
    explicit Managed() : ConcreteT() {}
```

```
#define CTOR( z, n, _) \
  template< BOOST_PP_ENUM_PARAMS( \
            BOOST_PP_INC(n), typename A) > \
  explicit Managed(BOOST_PP_ENUM_BINARY_PARAMS \
                  (BOOST_PP_INC(n), \
                  const A, & p)) \
            : ConcreteT(BOOST_PP_ENUM_PARAMS( \
                        BOOST_PP_INC(n), p)) {} \
    /**/
    BOOST_PP_REPEAT(MAX_ARITY, CTOR, ~)
#undef CTOR

  virtual ~Managed() {}
  virtual void ForceUseOfManaged(const
       ManagedBase::OnlyAvailableToManaged&)
       const {}
public:
  static InterfacePtr Create() {
    return SmartPtrPolicyTraits< SmartPtrT,
                                 InterfacePtr,
                                 ConcretePtr,
                                 ConcreteT >
              ::Initialise(new Managed);
  }

#define CREATE(z, n,) \
  template< BOOST_PP_ENUM_PARAMS( \
            BOOST_PP_INC(n), typename A) > \
  static InterfacePtr \
  Create(BOOST_PP_ENUM_BINARY_PARAMS( \
         BOOST_PP_INC(n), \
         const A, &p)) { \
    return SmartPtrPolicyTraits< SmartPtrT, \
                                 InterfacePtr, \
                                 ConcretePtr, \
                                 ConcreteT > \
            ::Initialise(new \
                Managed(BOOST_PP_ENUM_PARAMS( \
                        BOOST_PP_INC(n), p))); \
  } \
  /**/
  BOOST_PP_REPEAT(MAX_ARITY, CREATE, ~)
#undef CREATE
};
```

The above code tells the C++ pre-processor to generate `MAX_ARITY` constructor and Create function pairs. For more information on exactly how it does this, consult the boost `PreProcessor` library documentation on the Boost [boost] website.

There is no need to choose between either the constructor and `Create` function pairs solution or the partial template specialisation solution. Both solutions coexist quite happily together. The compiler will choose the partial template specialisation solution over the templated constructor and `Create` function pairs solution. Therefore you should use the templated constructor and `Create` function pairs solution by default, when the constructor parameters of your object are passed by const reference and write a partial template specialisation when your object requires constructor parameters to be passed by something other than const.

Finally we are at the stage where instances of our managed object can be created. The syntax used to create a managed object can be greatly simplified using another typedef:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
          ManagedConnection;
  ManagedConnection::InterfacePtr
      pConnection = ManagedConnection::Create(
                        "Connection String");
      pConnection->ExecuteSQL(
                        "SELECT * FROM MyTable");
}
```

Now that we have our managed object, we should check that it behaves in the way we expect. We have already seen that it can be created on the heap via the factory method; let's see if it can be created on the heap using `new`:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
          ManagedConnection;
  ManagedConnection::InterfacePtr
    pConnection
      = new ManagedConnection(
                "Connection String");
  ...
}
```

No. The compiler gives an error stating that it cannot access `operator new` and `operator delete` in `ManagedConnection` as they are private. So far so good; What about creating the managed object on the stack:

```
int main() {
  typedef Managed< Connection,
                   IConnection,
                   boost::shared_ptr >
          ManagedConnection;
  ManagedConnection connection(
          "Connection String");
}
```

No, this doesn't work either. The compiler gives an error, stating that it cannot access the private constructor and destructor in `ManagedConnection`. Ok, what about creating the Connection object directly on the heap:

```
int main() {
  IConnection* pIConnection =
         new Connection("Connection String");
}
```

No, this doesn't work. The compiler complains that it cannot instantiate `Connection` as it is an abstract class and that it cannot access `Connection`'s `new` and `delete` operators as they are private. What about creating `Connection` directly on the stack?

```
int main() {
  Connection
      connection("Connection String");
}
```

This final test doesn't work either. The compiler complains that it cannot create `Connection` as it is an abstract class. Making `Connection` static or a class member does not work either, for the same reason.

These five tests demonstrate that our managed object works as expected.

Finally

In this article I have discussed how to use smart pointers to prevent memory leaks and ways of forcing objects to be created on the heap only, while also preventing the possibility of a memory leak.

I expect this technique to be useful not only to library writers but also to general application writers who are concerned about possible misuse of their objects by other people.

I also hope this will appeal as a viable alternative to people thinking about Microsoft's Managed C++ as a solution to their memory leak problems.

Acknowledgments

Thanks to: James Slaughter, Adrian Fagg, Phil Nash, Mark Radford, Kevlin Henney, Allan Kelly, Mikael Kilpelainen, Jennifer Hart, Paul Mensonides, Tim Pushman

References:

[Aeryn] Aeryn: http://www.paulgrenyer.co.uk/aeryn/ (http://www.paulgrenyer.co.uk/aeryn/)

[Gamma-] *Design Patterns: elements of reusable object-oriented software* , Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, ISBN 0201633612

[Henney1] The method of creating objects at the highest level and then passing some form of reference around has come to be known in some circles as Parameterise from Above and is a much talked about work in progress by Kevlin Henney: http://www.java.no/web/moter/javazone03/presentations/KevlinHenney/Programmer_s%20Dozen.pdf (http://www.java.no/web/moter/javazone03/presentations/KevlinHenney/Programmer_s%20Dozen.pdf)

[Henney2] *Reference Counting*, Kevlin Henney: http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ReferenceAccounting.pdf (http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ReferenceAccounting.pdf)

[Henney3] *Factory and Disposal Methods*, Kevlin Henney: http://www.two-sdg.demon.co.uk/curbralan/papers/vikingplop/FactoryAndDisposalMethods.pdf (http://www.two-sdg.demon.co.uk/curbralan/papers/vikingplop/FactoryAndDisposalMethods.pdf)

[Henney4] *Executing Around Sequences*, Kevlin Henney: http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf (http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf)

[Meyers] *More Effective C++*, Scott Meyers, Addison Wesley, ISBN 020163371X

[gcc] gcc 3.2.3: http://gcc.gnu.org/ (http://gcc.gnu.org/)

[boost] Boost: www.boost.org/ (http://www.boost.org/)

[cuj] CUJ: http://www.cuj.com/ (http://www.cuj.com/)

[Myers] *Traits: a new and useful template technique*, Nathan C. Myers: http://www.cantrip.org/traits.html (http://www.cantrip.org/traits.html)

[Vandervoorde-] *C++ Templates: The Complete Guide*, David Vandervoorde, Nicolai M. Josuttis, Addison Wesley, ISBN 0201734842

[Sutter] *Befriending Templates*, Herb Sutter, http://www.cuj.com/documents/s=8244/cujcexp2101sutter/ (http://www.cuj.com/documents/s=8244/cujcexp2101sutter/)

[Meyers2] *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, Addison Wesley, ISBN 0201924889

[Henney5] *Data Abstraction and Heterarchy*, Kevlin Henney, http://www.cuj.com/documents/s=7992/cujcexp1908henney/ (http://www.cuj.com/documents/s=7992/cujcexp1908henney/)